Hardware code generation from dataflow programs

Nicolas Siret, Matthieu Wipliez, Jean Francois Nezan, Aimad Rhatay

HARDWARE CODE GENERATION FROM DATAFLOW PROGRAMS

Nicolas Siret[1], *Matthieu Wipliez*[2], *Jean-François Nezan*[2], *Aimad Rhatay*[1]

[1] Lead Tech Design, F-35700 Rennes, France
[2] IETR, INSA Rennes, F-35043, Rennes, France

ABSTRACT

The elaboration of new systems on embedded targets is becoming more and more complex. In particular, multimedia devices are now implemented using mixed hardware and software architecture, which improve the computational power but also increase the design complexity and the time to market. New design flows have been developed to help designers in the development of complex architecture. These design flows are often based on the use of languages with a higher level of abstraction. RVC-CAL is a dataflow programming language which provides the good features in this context. An RVC-CAL dataflow program can be compiled to various target software languages (e.g. C, Java, LLVM) with the Open RVC-CAL Compiler (Orcc). In this paper, we will present a new hardware code generator that generates a high-quality portable VHDL code with hierarchical architecture from a RVC-CAL dataflow program in a matter of seconds. The paper explains the underlying principles of the hardware code generator, and presents the results obtained from an Inverse DCT described as an RVC-CAL dataflow program.

Index Terms— RVC-CAL, Code generation, Hardware Synthesis, VHDL.

1. INTRODUCTION

New embedded systems, such as mobile phones, are currently offering more and more complex interactive contents, e.g. wireless telephony, multimedia applications, Internet surfing and Global Positioning Systems. To cope with the demand of new and complex contents, technological evolution must enable the integration of innovative designs in new System on Chip (SoC). These designs, which are usually made up of hardware IPs, embedded processors and software IPs, improve the computational power but increase the time needed to market the products and the related Research and Development cost.

New design flows have been developed to help designers by offering them adequate methods and functional hardware and software IPs. These design flows encourage the use of languages with a higher level of abstraction and the generation of hardware and software codes [1, 2] from abstract designs. Thus, companies such as Cadence[1] or Synopsys[2] provide their own design flows and code generators from SystemC designs. However, these tools are expensive and complex to use, especially when compared to mainstream hardware synthesizers.

Dataflow programming is an innovative method which enables new and complex architecture to be designed at a high level of abstraction. This method facilitates programming because it allows usual low-level difficulties, such as cycle accuracy and embedded memory management, to be abstracted. A dataflow-based design flow requires adequate tools: a simulator, a debugger and a multi-target code generator. Our purpose is to optimize the potential of the dataflow language RVC-CAL, by providing these tools.

In this paper, we focus on the hardware code generation of the multi-target code generator called Open RVC-CAL Compiler (Orcc). Orcc is an open source compiler that can generate code in various languages (e.g. VHDL, C, Java, LLVM) using a dataflow-oriented Intermediate Representation with simple semantics. This paper introduces the Orcc hardware code generator and the method to generate a readable, independent, target and an optimized VHDL code. The proposed approach is comparable to an existing RVC-CAL code generator called CAL2HDL. Results are validated on a subset of a MPEG-4 decoder case study, namely the IDCT provided by the abstract RVC-CAL design.

In section II the background of the research is introduced; section III presents the Orcc front-end and the intermediate representation; sections IV and V introduce the hardware back-end and the generated code; sections VI and VII outline practical results and perspectives for future works.

2. BACKGROUND

The increasing complexity of co-design architecture is an important problem that requires the use of new design flows. These flows encourage the use of hardware and software code generators to cope with the difficulty of programming at various levels of abstraction. This section presents both the usual code generators and our approach.

[1] C-to-silicon compiler
[2] Synphony High Level Synthesis from Language and Model-based Design

2.1. Related work

The main purpose of code generators is to allow designers to program complex designs at a single level of abstraction with one language, usually SystemC. SystemC is a set of C++ classes and macros, generally described as a system-level modeling language with features for hardware descriptions. An event-driven simulation kernel provided with SystemC enables designers to simulate sequential and concurrent processes in a real-time environment. As a result, SystemC designs can be simulated and validated at a high level of abstraction using proprietary tools and later convert them into low level programs. The time required to market complex architecture can also be reduced using other SystemC features, e.g. architectural exploration, performance modeling, high-level synthesis-verification, etc.

SystemC to Verilog (or VHDL) translation requires a compliant SystemC model [3], a design file which defines the hardware constraints and sometimes technological libraries which define the hardware target. The SystemC to hardware translator proceeds in two steps: (1) parsing of the input model to an Intermediate Representation (IR) and (2) generating a hardware description from the IR. The parsing process detects the instances, the Inputs/Outputs (I/Os) and the code, and prints them in intermediate files. Various transformations are performed during this process, e.g. blocking and non-blocking assignments to signals, management of arrays, etc. Once resolved, the intermediate code is directly printed into Verilog using the design file and the technological libraries (if available) to set hardware directives.

Today, results presented by code generators suppliers show benefits in terms of time to design, RTL quality and engineering effort. Despite these advantages, several difficulties are still present. The major one is the considerable number of possibilities offered by SystemC which requires the designers to make multiple choices on specific elements (e.g. mapping, technological target., etc.). As a result, highly skilled individuals are necessary to precisely define the application, to make decisions on the multiple choices offered, to program and to validate the application in SystemC.

To avoid the difficulties encountered with SystemC, other code generators have been developed. They generate hardware or software code from common language (C), abstract models (UML) or graphical programming models (Ptolemy). However, they are always less developed, less efficient or more complex to use.

2.2. Reconfigurable Video Coding

The Reconfigurable Video Coding (RVC) standard [4] aims at providing a system-level model of specification for existing and future MPEG standards. In this way, RVC improves the flexibility and the re-usability of codec features and facilitates the support of multiple codecs. RVC defines a reference language for the development of RVC applications called RVC-CAL, an actor-oriented dataflow language that enables the description of complex designs at a high level of abstraction. Contrary to SystemC, it imposes several rules (dataflow programing) to designers which reduce the complexity of new applications development.

RVC provides both a normative standard library of actors and a set of MPEG decoder descriptions expressed as networks of actors. RVC-CAL is supported by a simulator [5], an hardware code generator called CAL2HDL, and a multi-target code generator called Open RVC-CAL Compiler (Orcc) [6]. As shown in the example presented Fig. 1, an RVC-CAL dataflow program is built according to the Reconfigurable Video Coding (RVC) standard [7], in other words, as an abstract block diagram in which blocks define processing entities called actors or Functional Units (FUs), and connections between FUs represent dataflow.

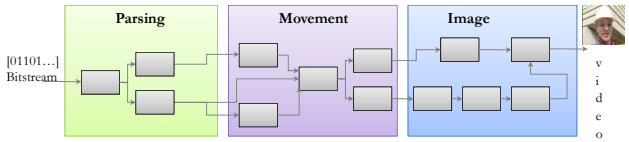

Fig. 1. Occurence of a MPEG-4 RVC decoder

An RVC-CAL actor communicates with other actors through input and output ports connected to FIFOs. Actors contain state variables, functions, procedures, and *actions* that may be identified by a tag. Tags are hierarchical: two actions tagged `a.b` and `a.c` respectively can be referred to the tag `a`. The only entry points of an actor are its actions; functions and procedures can only be called by an action. Additionally, an actor may have a Finite State Machine (FSM) whose transitions are tags (which means there can be several candidate actions per transition), as well as a set of priority inequalities that establish a partial order between tags (meaning that each priority inequality may concern sets of actions).

Figure 2 presents an actor named "Clip" that performs a *clipping* operation, which is traditionally done in video decoders after an inverse Discrete Cosinus Transform to clip the values of pixels at a given interval. The behavior expressed in this actor is as follows. When a boolean token arrives on the *SIGNED* port, the **read_signed** action fires, reads the token and stores it in the `sflag` state variable, and initializes a counter named ``count''. The `sflag` variable configures the clip as signed if `true` and unsigned otherwise. At this point only the **limit** action can fire, regardless of whether there are tokens on the *SIGNED* port, because ``count'' is not negative. Each time this action fires, it reads a token on *I*, decrements the counter, clips the value read to $[-255, 255]$ signed mode or to $[0, 255]$ unsigned mode, and outputs the clipped value on *O*. When `count` reaches -1, the cycle can start again.

Previous research [8] on CAL2HDL have shown profits in

```
actor Clip ()
  int(size=10) I, bool SIGNED ==> int(size=9) O :

  int(size=7) count := -1;
  bool sflag;

  read_signed: action SIGNED:[s] ==>
  guard count < 0
  do
    sflag := s;
    count := 63;
  end

  limit: action I:[i] ==> O:[ if i > 255 then 255
    else if i < min then min else i end
    end ]
  var
    int min = if sflag then -255 else 0 end
  do
    count := count - 1;
  end

  priority
    read_signed > limit;
  end

end
```

Fig. 2. The `Clip` actor in RVC-CAL.

terms of performance and RTL quality. However, CAL2HDL is not a satisfactory solution for various reasons. First of all, it generates a design which is platform-dependent and can only be used on Xilinx FPGAs. Then, the time to generate the Verilog design increases dramatically with the design complexity, i.e. a few seconds for a simple design like an *IDCT* and more than twenty minutes for a complete *MPEG-4 Simple Profile decoder*. Moreover, the design hierarchy is lost during the parsing operation, the final design is flattened and the generated Verilog is unreadable, which makes low-level debugging nearly impossible. Finally, the code generator itself is complex and difficult to update. To overcome these limitations, our approach aims at offering a new and effective hardware code generator by adding a new back-end to the Orcc Compiler.

2.3. Open RVC-CAL Compiler (Orcc)

Orcc is an open-source compiler written in Java and available as a feature for the Eclipse environment. It can generate code in various software target languages (e.g. C [9], Java, LLVM) from RVC-CAL dataflow programs. As shown in Fig. 3, Orcc compiles a dataflow program in two steps: (1) the front-end transforms the source program into an Intermediate Representation (IR), and (2) a back-end for a given language transforms the IR into this language.

3. INTERMEDIATE REPRESENTATION OF RVC-CAL ACTORS

In the context of co-design and heterogeneous computing more generally, it is advisable to be able to compile RVC-

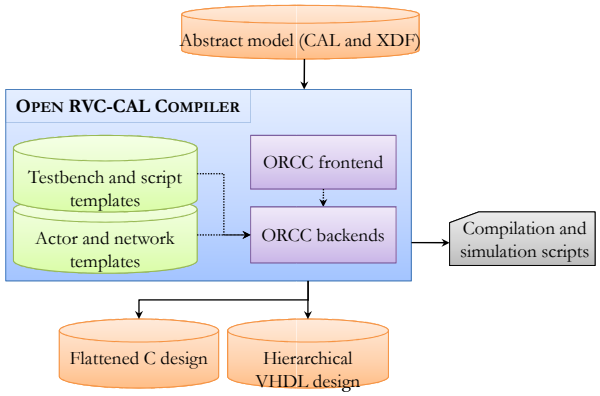

Fig. 3. Open RVC-CAL Compiler design flow

CAL actors into several more traditional languages, hardware and software alike (C, Java, VHDL, etc.). Compiling actors into any of these languages requires the code of actors to undergo several transformations. Indeed, several high-level functional constructs in RVC-CAL have no direct equivalent in lower-level languages like C and VHDL. RVC-CAL does not distinguish between assignments to local and state variables, but Hardware Description Languages (HDLs) generally do. The language also has concepts that are orthogonal to some Abstract models, e.g. Finite State Machines (FSMs) must be expressed using specific constructs (such as `gotos` or `switchs`) in most software languages, while FSMs are first-class citizens of mainstream HDLs.

The number of target-specific transformations that must be programmed to compile actors into different target languages can be minimized by using an Intermediate Representation (IR) of actors. This IR would express the same structural and semantic information as RVC-CAL using constructs available in virtually any language so as to ease the translation into other languages.

The Open RVC-CAL Compiler compiles a dataflow program by first compiling the RVC-CAL actors to an IR, hereinafter named "Orcc IR", and then compiling the IR into a given target language with target-specific transformations. The Orcc IR keeps much of the structural information of RVC-CAL, although not necessarily in the same way:

- State variables, functions, and procedures in the source code are state variables and functions in the IR.
- Each RVC-CAL action is transformed into two functions in the IR, one that contains the action's schedule test, and another that contains the body of the action.
- The partial order of actions expressed by priorities becomes a total order of actions in the IR (see [9] for more details).
- The source FSM is translated into an FSM in the IR,

with the difference that the latter has expanded references to actions.

The semantic information expressed by functions, procedures, and actions is transformed using a lower-level language with the following semantics:

- The high-level RVC-CAL functional expressions containing function calls, conditionals or list generators are translated into an equivalent lower-level IR expression.
- The assignment statement is differentiated into assignments to local variables and load/store to memory operations.
- Functional tests and list generators become imperative statements.

Orcc contains a Java implementation of the IR along with common transformations (copy propagation, dead variable removal, etc.)

4. HARDWARE CODE GENERATION

This section presents our method to generate a complete hierarchical hardware description from an RVC-CAL dataflow program. Our code generator transforms the target-agnostic IR of actors to an IR that respects a valid subset of VHDL semantics, from which VHDL code is printed. Each network is translated into a VHDL description. Finally, benchmark scripts are generated for each actor and network. The method described in this paper are implemented in the Orcc tool as the VHDL back-end.

4.1. Transformation of the IR

The transformation first eliminates dead code and dead variables, if any. Then, it renames the variables, e.g.

- **temp**_*variable* $\xrightarrow{\text{VHDL}}$ **instance_process**_*variable*.

Finally, conditional actions, types of variables and CAL expressions are transformed from the IR shape into a VHDL shape, e.g.

- **if** (*condition*) $\xrightarrow{\text{VHDL}}$ **if** (*condition* = *'1'*) **then**
- **bool** *variable* $\xrightarrow{\text{VHDL}}$ *variable* : **std_logic**
- **list** *(variable)* $\xrightarrow{\text{VHDL}}$ **array***(size(list))* of **type(variable)**
- *variable* = **if** (*condition*) $\xrightarrow{\text{VHDL}}$ **if**(*condition*) **then** *variable* = *'1'* **else** *variable* = *'0'* **end if**.

4.2. Printing Actors and Networks

Hardware code for actors and networks is generated from the IR of actors and networks with templates. A template contains chunks of text interleaved with code or references to data. Using templates is more flexible compared with other programming approaches such as `println`-based ones. Using templates enables the model-view separation. The syntax of the generated code is thus independent from the model of the code so that the structure, the syntax or the layout of the generated code can be changed more easily. As presented in Fig. 4, a template defines all the necessary VHDL instructions; it also contains the header (e.g. libraries) and the footer (e.g. note) which are usually inserted in each VHDL program. We use the StringTemplate template engine to print code using templates.

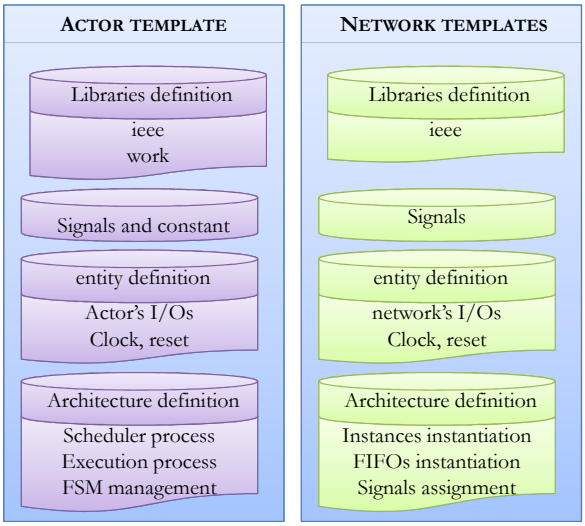

Fig. 4. Architecture of the actor and networ templates

The actor template defines the libraries, the entities, the signals, the architectures and the processes of the generated VHDL code. The libraries printed are the *ieee* library (`std_logic` and `numeric_std` packages) in addition to the *work* library (`orcc_package`). The inputs and outputs of the original actor can be retrieved in corresponding VHDL entity but additional control signals are added to the VHDL entity : *send* and *acknowledgment* for each actor's input and *ready*, *write* for each actor's output. The VHDL architecture is printed according to the core of the RVC-CAL action. RVC-CAL constants are translated into constants in VHDL, global variables into signals, local variables into variables and lists into array of signals, or array of constants. The process in the VHDL code contains an FSM if the original actor contains a FSM.

Like the actor template, the network template holds all the necessary instructions to print the networks. The libraries printed are the *ieee* library and the *work* library. The net-

work entity is printed using the same network's I/Os and two additional inputs (clock and reset). The VHDL architecture generated from a network instantiates actors, sub-networks, and FIFOs between all the instances, as shown in Fig. 5.

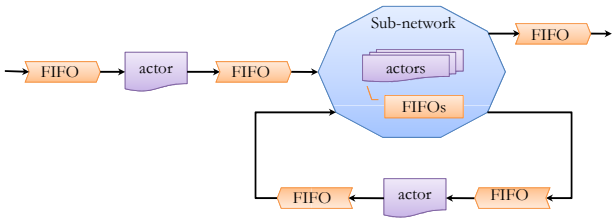

Fig. 5. Schematic representation of a network printed with Orcc-VHDL

4.3. Printing Testbenches and Scripts

In addition to hardware code, the code generator generates one test bench file for each actor and network, as well as a simulation script. This script can be executed by a simulator like Modelsim in order to launch a simulation of the whole network with a single click. The test bench template is written in the same way as the actor and the network template. As shown in Algorithm 1, it also contains the actor or the network instantiation, but also the code to test the VHDL entity.

Algorithm 1 Structure of the testbench template

```
1:  libraries definition -- both ieee and work
2:
3:  entity definition -- entity is empty
4:
5:  architecture
6:  if (instance is network) then
7:      print_Network_Instantiation()
8:      print_NetworkInput_Testbench()
9:      print_NetworkOutput_Testbench()
10: else
11:     print_Actor_Instantiation()
12:     print_ActorInput_Testbench()
13:     print_ActorOutput_Testbench()
14: end if
15: end architecture
```

The testbench template is built so as to generate a code which requires only stimulus text files to be used, which makes generated test benches easy to use with any VHDL simulator. As a matter of fact, we have been using the test benches to validate the back-end by comparing the inputs and outputs of the VHDL code and the reference inputs and outputs of the C code generated with orcc from the original RVC-CAL program.

The generated scripts are also printed using templates. When processing a design, VHDL compilers and synthesizers load and instantiate components in a depth-first order. As a result, our script must compile actors and networks in the same order: actors first and then networks that contain only actors, followed by networks that contain these networks, and so on.

5. STRUCTURE OF THE GENERATED CODE

The quality of the generated code is another major objective of our approach. To this end, the code must be generic, VHDL93 compliant and understandable in order to be used and modified by designers. In this section, we present the characteristics of the generated code.

5.1. Structure of the generated code from an actor

As presented in Fig. 6, the architecture of the VHDL generated instances is similar to the hand-coded one: the VHDL entity contains the input and output ports; the RVC- CAL global variables are transformed into VHDL signals; the VHDL architecture contains the core of the process and the local variables are used only inside a process in order to avoid the use of memory. The entity of an actor is defined by the I/Os of the RVC- CAL model in addition to a clock and a reset port. The architecture of an actor is always made up of two parts: a combinatorial process and a sequential process.

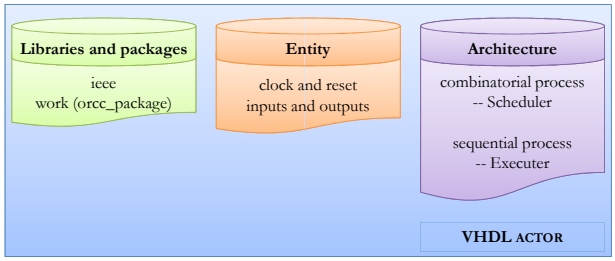

Fig. 6. Architecture of a VHDL actor

The combinatorial process tests if it is possible to execute an action (see section 3), e.g. if necessary data are present in the input port of the actor. RVC-CAL inputs of an actor constitutes the sensitivity list of the process so that the process is activated only if those inputs are modified. The tested signals are are associated a *count* value which defines the iteration number of an action. The sequential process contains the core of an actor. In fact, each action of an RVC-CAL actor is printed in the process and the code operation is kept. As shown in Algorithm 2, a single action is executed each cycle; this is due to RVC-CAL global variables which are transformed into VHDL signals so that they are updated on a clock cycle.

Algorithm 2 Test which process can be executed

```
1: architecture
2: -- local variables
3:
4: process(sensitivity list)
5: all_actions_wait;
6: if (action_1 is executable) then
7:    action_1_go
8: else
9:    if (action_2 is executable) then
10:      action_2_go
11:   else
12:      if (action_3 is executable) then
13:         action_3_go
14:      else
15:         if (action_n is executable) then
16:            -- etc
17:         end if
18:      end if
19:   end if
20: end if
21: end process
22: end architecture
```

Our method allows a high throughput of one data per cycle using local variables to execute the compute operations, and global variables to memorize necessary operators in the action. In fact, the instantaneously update of local variables enables us to manage more than one operation per cycle. Since most of the operations are arithmetic operations, the local variables are defined as integer with a size defined on the RVC- CAL model (32 bits if no size is given). Special instructions needed to process algorithms are defined in a package called *orcc_package*. Precisely, the package contains the following functions:

- *binary_and*, *binary_or* and *binary_xor* $\rightarrow$ Process a logical *and*, *or* and *xor* operations respectively between two operators.
- *binary_not* $\rightarrow$ Processes a logical *not* operation on an operator.
- *div* $\rightarrow$ Processes a *division* between two operators.
- *get_mod* $\rightarrow$ Processes a *modulo* operation between two operators.
- *shift_left* and *shift_right* $\rightarrow$ Process a binary *shift left* and *shift right* operation respectively on an operator.
- *concatenation* $\rightarrow$ Processes a *concatenation* between two operators.

5.2. Structure of the generated code from a network

The network instances are printed according to Fig. 5 presented in section 4. The code is made up of four parts: the definition of usual libraries, the entity definition, the signal declaration and the entity instantiations. The actors, sub-networks and FIFOs instantiations are printed using the *instance_n : entity library.instance* shape. This allows reducing the code size and testing the presence of all instances during the compilation in addition to be compliant with the VHDL93 standard.

Once optimized, the actors can operate at high frequency, i.e. more than 150MHz on a usual FPGA. However, performance of a design is restrained by the inter-connection between actors. In other words, FIFOs limit the maximal operating frequency and the logical use. To ensure good performance while reducing the logical use, the FIFOs are fitted to requirements defined in the RVC-CAL model. Two kinds of FIFOs are used in a VHDL network: a high performance FIFOs which use a single register and regular FIFOs which use embedded memory.

Algorithm 3 Management of data transmission in FIFOs

```
1: if register is empty then
2:    if (data send from an actor n) then
3:       store the data from actor n
4:       send an acknowledgment to actor n
5:       send a ready to actor n +1
6:    end if
7: else
8:    if (data send from an actor n) then
9:       if (data load from an actor n + 1) then
10:         store the data from actor n
11:         send an acknowledgment to actor n
12:         send a ready to actor n +1
13:      else
14:         send a ready to actor n +1
15:      end if
16:   end if
17: end if
```

Hhigh performance FIFOs use an *arbiter* to manage the register and the actors. Thus, actors can process simultaneously without data loss because an action is executed when it has the permission from the FIFO's *arbiter*. As presented in Algorithm 3, permission is sent by the arbiter if restricted conditions are validated.

5.3. Structure of the generated code for the testbenchs

To facilitate the debug of a RVC-CAL model, a test bench is generated for each instance (i.e. actors and networks). As presented in Fig. 7, the testbench is built so it is as simple as possible to use. It requires input files which contain the data to be tested and automatically reports the errors between the outputs of the instance tested and the reference files.

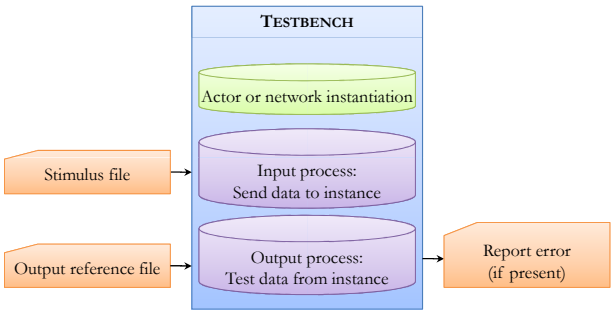

Fig. 7. Automatical test of an instance using a stimulus file and a reference file

The input and output processes are made up of FSM (i.e. one FSM per input/output) which read the stimulus file corresponding to the port of an actor and send/test the data. When the data tested from the actor and the stimulus file are not equal, an error is automatically printed in the console window using the dedicated instructions: *assert*, *report* and *severity*.

6. RESULTS

Results are provided with a classical IDCT design made up of five actors. The generated code has not been optimized in any way, and the results presented are obtained using common simulators and synthesizers.

6.1. Simulation and Functional Verification

The design was tested on a MPEG-4 media video (CIF resolution) with the VHDL backend. Even if the MPEG-4 RVC-CAL can be entirely generated, the simulations and synthesis are restricted to the IDCT network. In fact, the IDCT is made up of actors which use the majority of the RVC-CAL code possibilities while remaining short. As introduced in Fig. 8, the VHDL design keeps the same ratio between input and output tokens (labeled throughput on the figure) as the RVC-CAL model. In terms of latency, eleven cycles are required between the first data received by the network and the first data sent.

Actor	CAL throughput	VHDL throughput (per cycle)
Scale	1	1
Combine	1	1
ShuffleFly	0,5	0,5
Shuffle	0,5	0,5
Final	1	1
Total	0,25	0,25

Fig. 8. Ratio between input tokens fired and output tokens produced (abstract throughput)

The classical arithmetic operations, the FSM and the memory (the list in RVC-CAL) work as well as the FIFOs which ensure that no data is lost in transmission. The halved throughput of *shuffleFly* and *shuffle* is due to their FSM, which first stores the input data and then sends the output.

An efficient and understandable VHDL/C design can be generated in a few seconds from complex RVC-CAL models, thus, the objectives of our approach are achieved. The simulation results are good and similar to those obtained with the existing hardware code generators. Contrary to CAL2HDL (see, section II), the time taken to generate is only a few seconds, regardless of the design complexity and the generated code is always understandable regardless of an actor's complexity and the model's hierarchy is kept.

6.2. Hardware Synthesis and Occupation

The IDCT model case study is an interesting way to evaluate the interest of our approach. Indeed, it is a design made up of different kinds of actors that use the majority of the instructions, e.g. arithmetic operations, lists that are converted into memory type (RAM or ROM) and state machines that are converted into VHDL FSM. Results presented on Fig. 10, were obtained using Xilinx ISE on a Xilinx Virtex 4 platform (ML402 evaluation platform) for it is a CAL2HDL compliant platform.

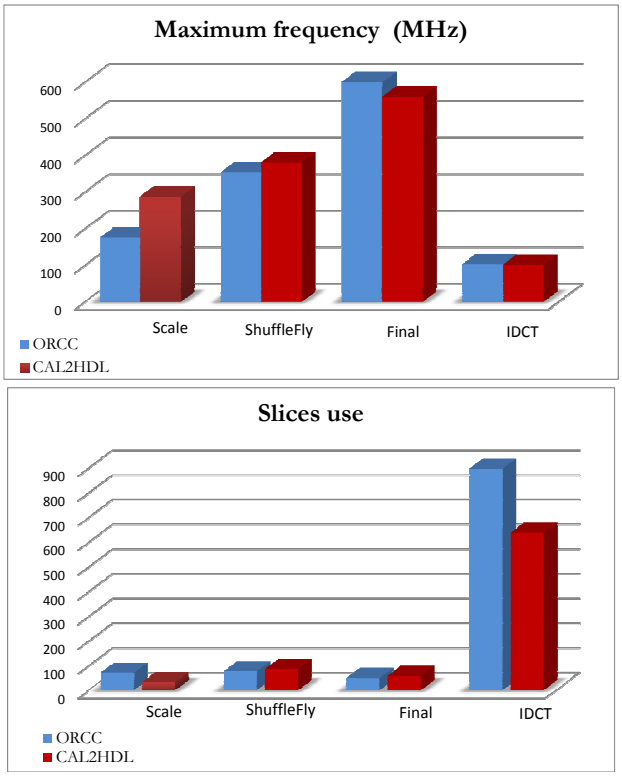

Fig. 9. Design performances

Comparing the results highlights a profit in terms of slice use for the actors and a profit in terms of frequency for the design with the Orcc generator. The increase of the slice use compared to CAL2HDL is due to the FIFO coding. In fact, we chose to focus on the performance of the design, coding a quick FIFO, despite the fact that it uses more slices. The FIFO is hand-coded, which means it could be changed to be smaller but slower if lower slice consumption is desired. The slice use will also be reduced in future stages of development by improving the management of complex hardware operations e.g. memory management.

7. CONCLUSION AND FUTURE WORK

This paper has presented a method to generate a high-performance, target independent VHDL code from a dataflow program expressed using the RVC-CAL language. This enables a design flow that uses Orcc-VHDL as the code generator. This design flow enhances the reuse of IPs, improves the management of hardware/software interfaces and facilitates the Research and Development of co-design architecture. Moreover, the generated code is target-independent and optimized so as to offer good performance while reducing the logical use.

Results show that the IDCT implemented in hardware is functional and provides good results in terms of throughput (i.e. until one data per cycle) and operating frequency (i.e. more than 100MHz). In addition, the various specific instructions (e.g. list) and elements (implicit FIFOs) are implemented through dedicated hardware structures (e.g. RAM, ROM, registers) while the code architecture is adapted using processes (both combinatorial and sequential), functions and procedures. Compared to the state of art, the use of an innovative intermediate representation allows the back-end to fit a specific application or a specific target. Finally, as presented in the paper, the hardware code is efficient, scalable and generated in few seconds whatever the design complexity.

Several optimizations are studied to improve the possibilities offered by the Orcc-VHDL back-end. In the case of co-design architecture, the IPs could be generated in hardware or software directly with the correct interface. It could be possible for the designers to develop and validate their entire application in CAL and to directly generate the co-design architecture. Another interesting area of future research involves managing the frequency of each actor depending on their throughput. The electrical consumption would be reduced adapting the frequency of various fields of actors (e.g. a frequency of f MHz for the slowest and frequency of f/n for the quickest with n = 2, 4, 8, etc). Finally, we are investigating the translation of higher-level RVC-CAL constructs such as for-loops and multi-token reads and writes into optimized low-level VHDL code.

References

[1] Wen Quan, "System-level co-design methodology based on platform design flow for system-on-chip," in *Computer-Aided Industrial Design and Conceptual Design, CAIDCD*, 2006.

[2] A. Kumar and P R. Panda, "Front-End Design Flows for Systems on Chip : An Embedded Tutorial," in *International Conference on VLSI Design, VLSID*, 2010.

[3] J. Castillo, P. Huerta, and J.I. Martnez, "An Open-Source Tool for SystemC to Verilog Automatic Translation," in *Latin American Applied Research*, 2007.

[4] M. Mattavelli, I. Amer, and M. Raulet, "The Reconfigurable Video Coding Standard [Standards in a Nutshell]," *Signal Processing Magazine*, vol. 27, no. 3, pp. 159 – 167, 2010.

[5] Shuvra S. Bhattacharyya, Gordon Brebner, Jörn W. Janneck, Johan Eker, Carl von Platen, Marco Mattavelli, and Mickaël Raulet, "OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems," *SIGARCH Comput. Archit. News*, vol. 36, no. 5, pp. 29–35, 2008.

[6] J.W. Janneck, M.Mattavelli, M.Raulet, and M.Wipliez, "Reconfigurable video coding: a stream programming approach to the specification of new video coding standards," in *Proceedings of the first annual ACM SIGMM conference on Multimedia systems, MMSys*, 2010.

[7] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet, "Overview of the MPEG Reconfigurable Video Coding Framework," *Springer journal of Signal Processing Systems. Special Issue on Reconfigurable Video Coding*, 2009.

[8] J.W. Janneck, I.D. Miller, D.B. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study," *Springer journal of Signal Processing Systems. Special Issue on Reconfigurable Video Coding*, 2009.

[9] Matthieu Wipliez, Ghislain Roquier, and Jean-Franois Nezan, " Software Code Generation for the RVC-CAL Language ," *Springer journal of Signal Processing Systems*, 2009.